Best practice in using OpenSSL to verify signature and device authentication

Contents

1. Some basic concepts
2. Practice OpenSSL in Device authentication

VC IVI Development Center Vietnam

Friday, Aug 20, 2021

Contents

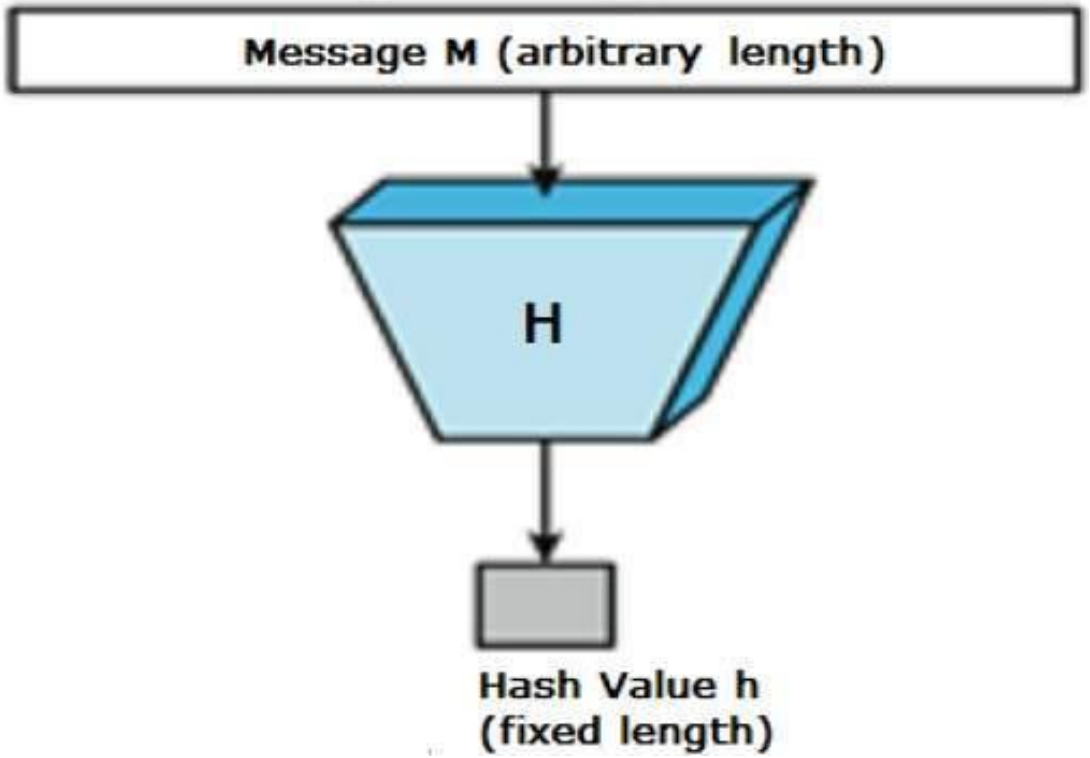
- Some basic concepts
- Practice OpenSSL in Device authentication

Some basic concepts

- Crypto hash function
- Encryption and Decryption
- Signature

What is crypto hash function ?

- A crypto hash function is a mathematical algorithm that maps data of arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function which is practically infeasible to invert or reverse the computation.



Some crypto hash algorithms – MD5

- MD5 was designed by Ronald Rivest in 1991 to replace an earlier hash function, MD4, and was specified in 1992 as RFC 1321.MD5 was designed by Ronald Rivest in 1991 to replace an earlier hash function, MD4, and was specified in 1992 as RFC 1321.
- MD5 produces a digest of 128 bits (16 bytes).

Some crypto hash algorithms – SHA1

- SHA-1 was developed as part of the U.S. Government's Capstone project. The original specification – now commonly called SHA-0 – of the algorithm was published in 1993 under the title Secure Hash Standard, FIPS PUB 180, by U.S. government standards agency NIST (National Institute of Standards and Technology).
- SHA-1 produces a hash digest of 160 bits (20 bytes).

Some crypto hash algorithms – SHA-2

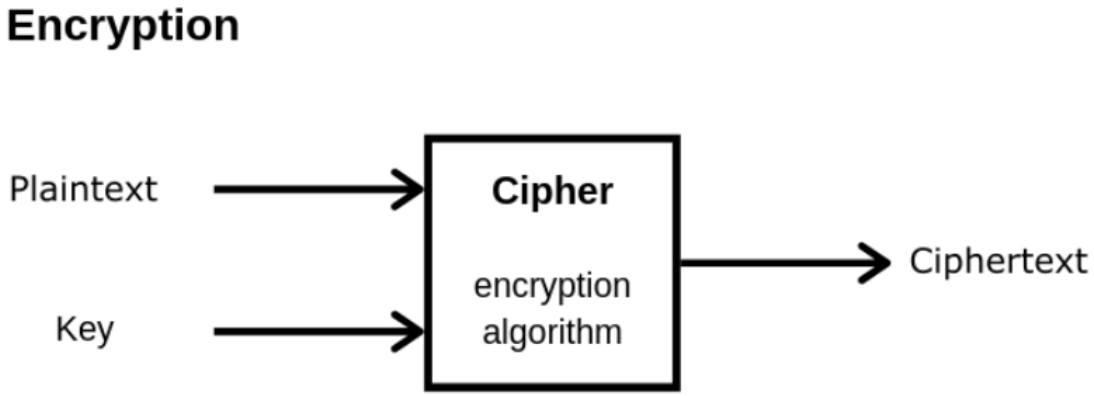
- SHA-2 (Secure Hash Algorithm 2) is a set of cryptographic hash functions designed by the United States National Security Agency (NSA), first published in 2001.
- SHA-2 basically consists of two hash algorithms: SHA-256 and SHA-512.
- SHA-224 is a variant of SHA-256 with different starting values and truncated output. SHA-384 and the lesser-known SHA-512/224 and SHA-512/256 are all variants of SHA-512.
- The output size in bits is given by the extension to the "SHA" name, so SHA-224 has an output size of 224 bits (28 bytes); SHA-256, 32 bytes; SHA-384, 48 bytes; and SHA-512, 64 bytes.

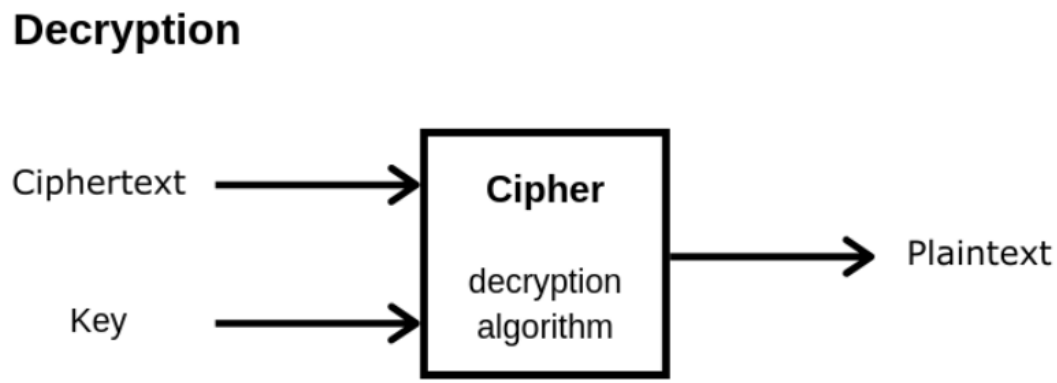
Some crypto hash algorithms – SHA-3

- SHA-3 (Secure Hash Algorithm 3) was released by NIST on August 5, 2015.
- SHA-3 provides the same output sizes as SHA-2: 224, 256, 384, and 512 bits.

Encryption and Decryption

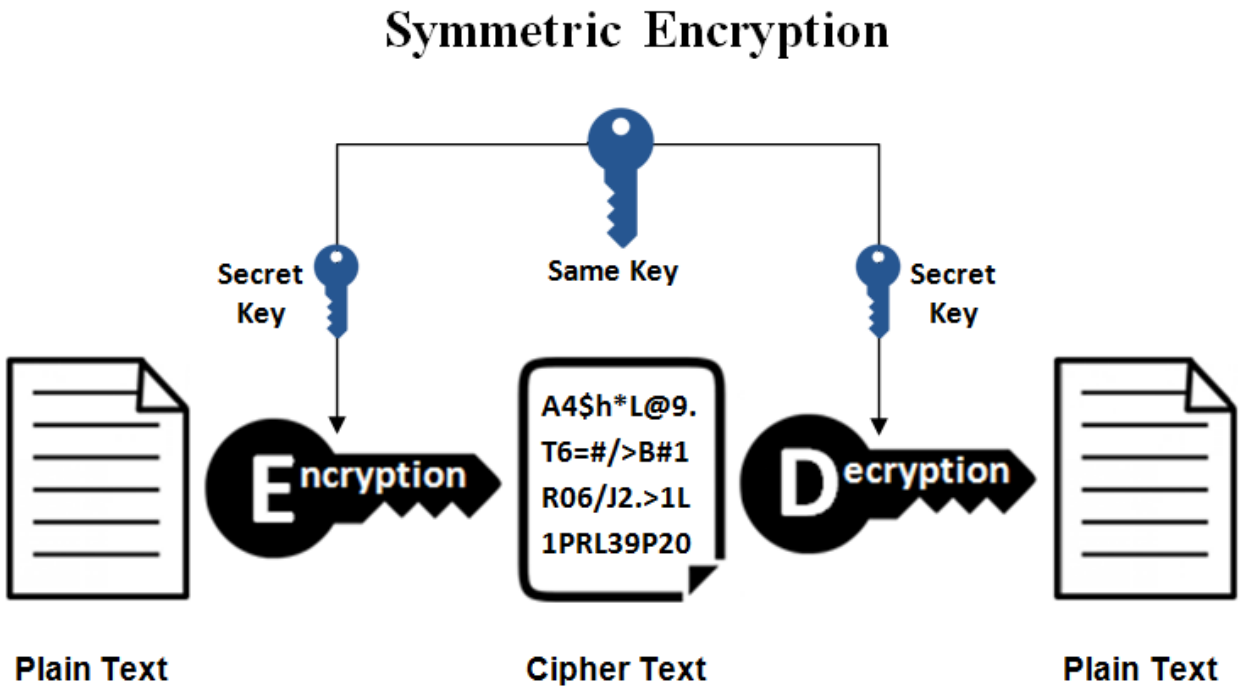
- Encryption is the process of translating plain text data (plaintext) into something that appears to be random and meaningless (ciphertext). Decryption is the process of converting ciphertext back to plaintext.
- Encryption algorithms can divide to 2 types: **Symmetric Encryption and Asymmetric Encryption.**





Encryption algorithms - Symmetric Encryption

- The symmetric encryption method, as the name implies, uses a single cryptographic key to encrypt and decrypt data.
- The use of a single key for both operations makes it a straightforward process, and hence it's called "symmetric."



Symmetric Encryption - DES (data encryption standard)

- DES was one of the most popular block symmetric ciphers. It was created in the early 1970s at IBM and adopted as a federal standard by NBS in 1976.
- DES is considered to be a well-designed and effective algorithm. However, just after its publication, many cryptographers believed that the size of its key is too small. At present, the 56-bit long key can be broken relatively cheaply, by using brute force attacks within a few days.

Symmetric Encryption - Triple DES (3DES)

- 3DES cipher is quite popular block symmetric cipher, created based on DES cipher. It applies the DES cipher algorithm three times to each data block.
- 3DES has a key length of 168 bits (three 56-bit DES keys) but due to the meet-in-the-middle attack, the effective security it provides is only 112 bits (because the third key is the same as the first).

Symmetric Encryption - AES

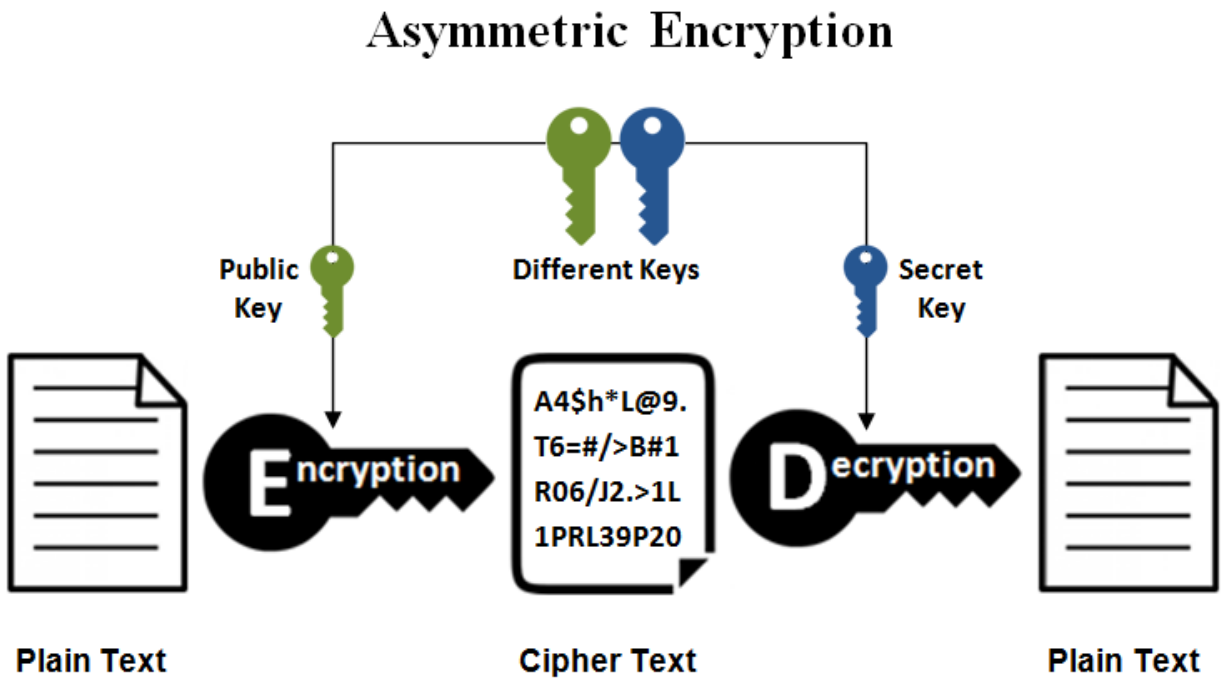
- AES is a modern block symmetric cipher, one of the most popular ciphers in the world. It was developed in 1997 by Vincent Rijmen and Joan Daemen, and later approved as a federal encryption standard in the United States in 2002.
- A secret key in AES, for both data encryption and decryption, may contain 128 or 192 or 256 bits. Based on the length of the key, a different number of encrypting cycles is performed.

Symmetric Encryption – RC4

- RC4 is a stream symmetric cipher. It operates by creating long keystream sequences and adding them to data bytes.
- RC4 encrypts data by adding it XOR byte by byte, one after the other, to keystream bytes. The whole RC4 algorithm is based on creating keystream bytes.
- Key length: up to 2048 bits

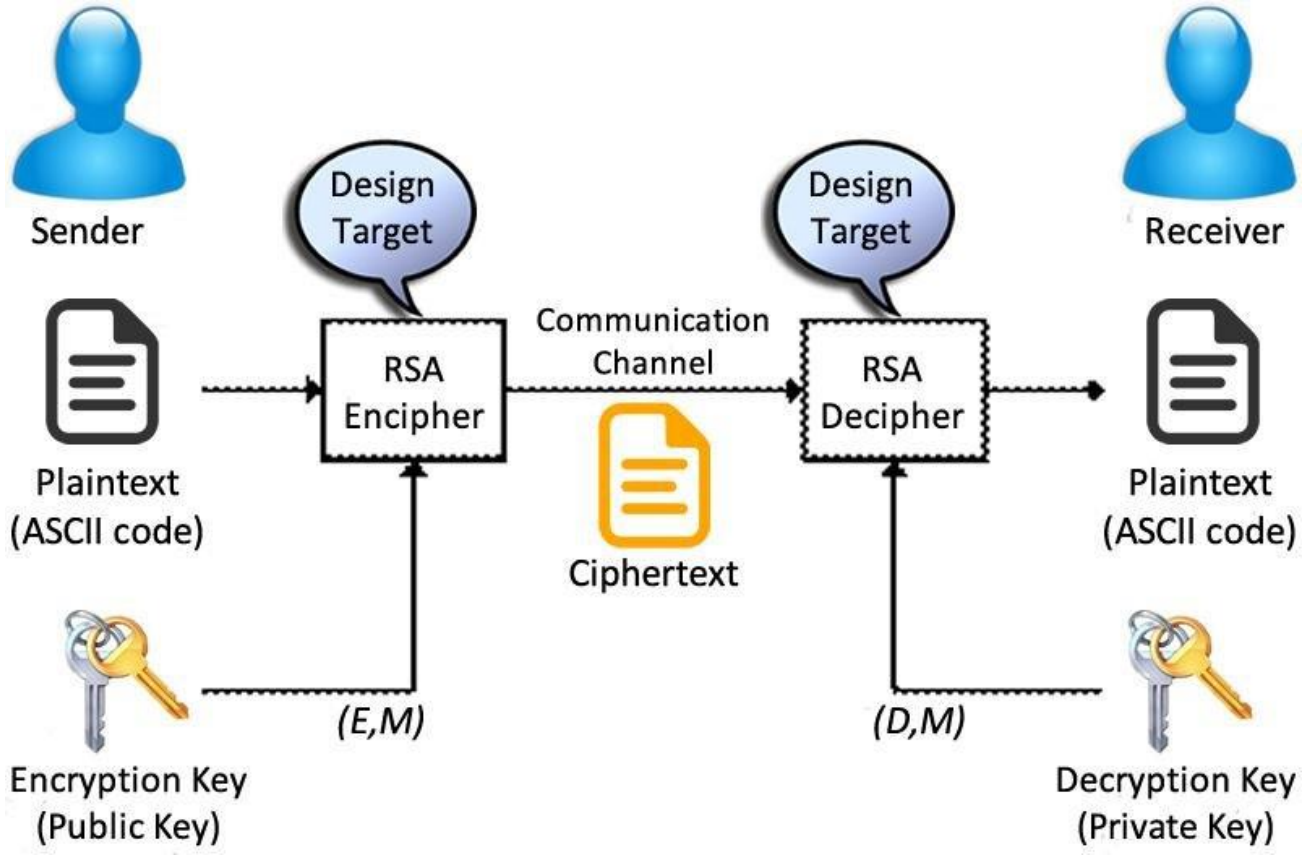
Encryption algorithms - Asymmetric Encryption

- Asymmetric encryption, in contrast to the symmetric encryption method, involves multiple keys for encryption and decryption of the data.
- Asymmetric encryption encompasses two distinct encryption keys that are mathematically related to each other. These are public key and private key.



Asymmetric Encryption - RSA

- RSA was designed by Ron Rivest, Adi Shamir and Leonard Adleman in 1977.
- RSA in one of the most popular algorithms with public key encryption. It can be used for either encryption of messages or for digital signatures.

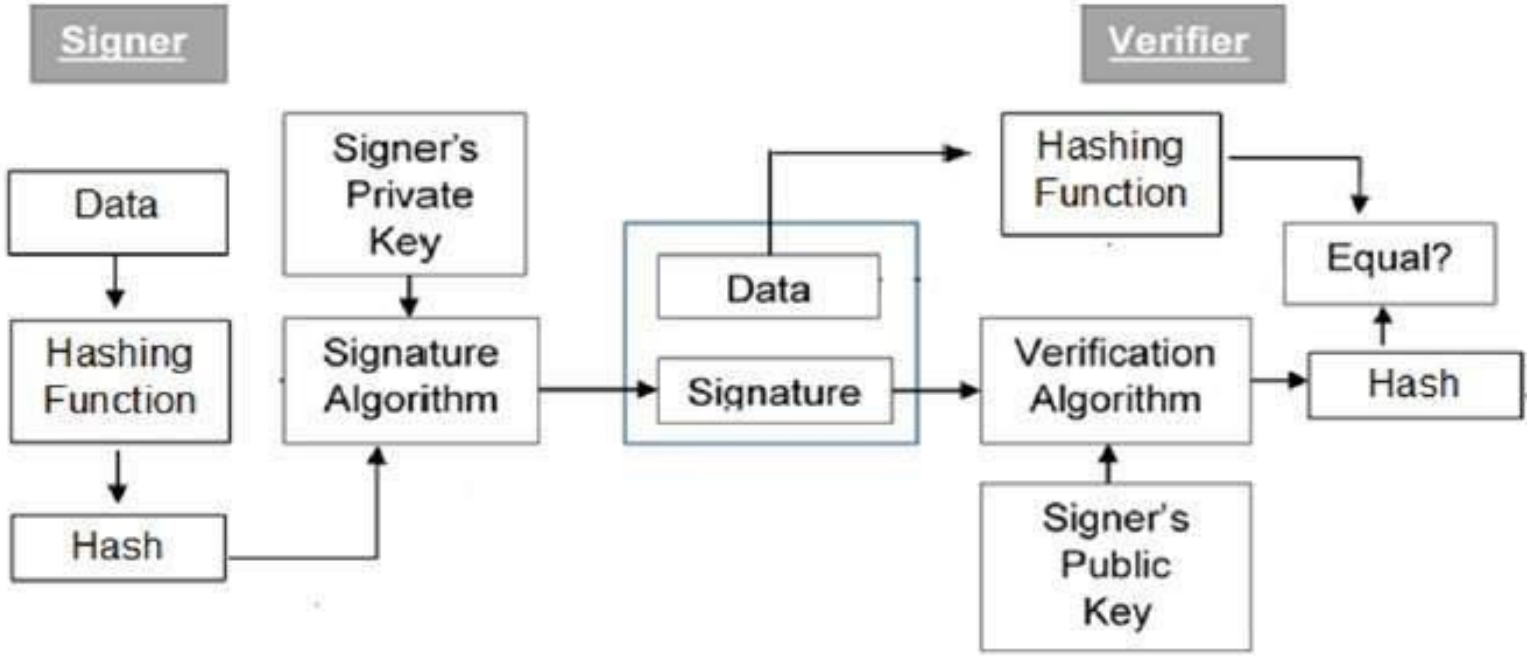


Asymmetric Encryption - ECC (Elliptic-curve)

- The Elliptic Curve Cryptography (ECC) is modern family of public-key cryptosystems, which is based on the algebraic structures of the elliptic curves over finite fields and on the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP).
- **ECC** implements all major capabilities of the asymmetric cryptosystems: **encryption**, **signatures** and **key exchange**.

What is signature ?

- A signature is a mathematical technique used to validate the authenticity and integrity of a message, software or digital document.
- Using a public key algorithm, such as RSA (Rivest-Shamir-Adleman) or ECDSA(Elliptic Curve Digital Signature Algorithm), two keys are generated, creating a mathematically linked pair of keys, one private and one public.



Practice OpenSSL in Device authentication

- What is OpenSSL and how to use in linux environment ?
- Using OpenSSL in device authentication.

What is OpenSSL ?

- OpenSSL is a robust and full-featured toolkit for Transport Layer Security(TLS) and Secure Socket Layer(SSL) protocols.
- It is also a general purpose cryptography library.
- OpenSSL is written in C programming language and relies on different cipher and algorithms to provide encryption.

Application Programming Interface of OpenSSL

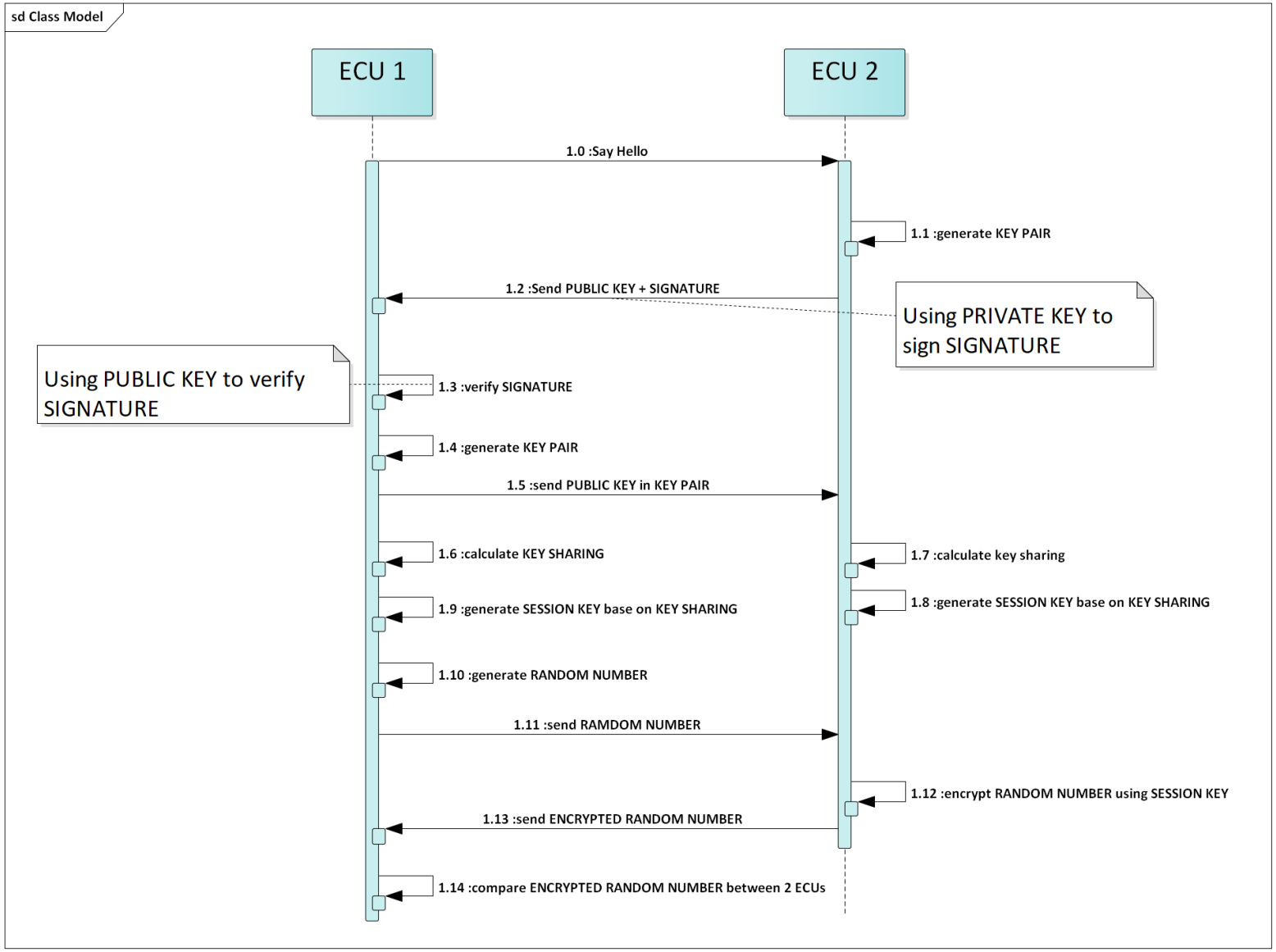
- libssl.a or libssl.so
  - Implementation of SSL_v2/3 & TLS_v1
- libcrypto.a or libcrypto.so
  - Ciphers (AES, DES, RC2/4, Blowfish, IDEA)
  - Digests (MD5, SHA-1, MDC2)
  - Public Keys (RSA, DSA, DH)
  - X509s (ASN.1 DER & PEM)
  - Others (BIO, BASE64)

How to install OpenSSL ?

- To use OpenSSL as command line. We can use command “sudo apt install openssl” to install.
- To use api of OpenSSL in our source code. We use command “sudo apt install libssl-dev”.

```
mai@mai-VirtualBox:~$ sudo apt install libssl-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  libssl-dev
0 upgraded, 1 newly installed, 0 to remove and 94 not upgraded.
Need to get 1.346 kB of archives.
After this operation, 7.203 kB of additional disk space will be used.
Get:1 http://vn.archive.ubuntu.com/ubuntu xenial-updates/main amd64 libssl-dev amd64 1.0.2g-1ubuntu4.19 [1.346 kB]
Fetched 1.346 kB in 9s (135 kB/s)
Selecting previously unselected package libssl-dev:amd64.
(Reading database ... 230104 files and directories currently installed.)
Preparing to unpack .../libssl-dev_1.0.2g-1ubuntu4.19_amd64.deb ...
Unpacking libssl-dev:amd64 (1.0.2g-1ubuntu4.19) ...
Setting up libssl-dev:amd64 (1.0.2g-1ubuntu4.19) ...
mai@mai-VirtualBox:~$
```

Device authentication sequence

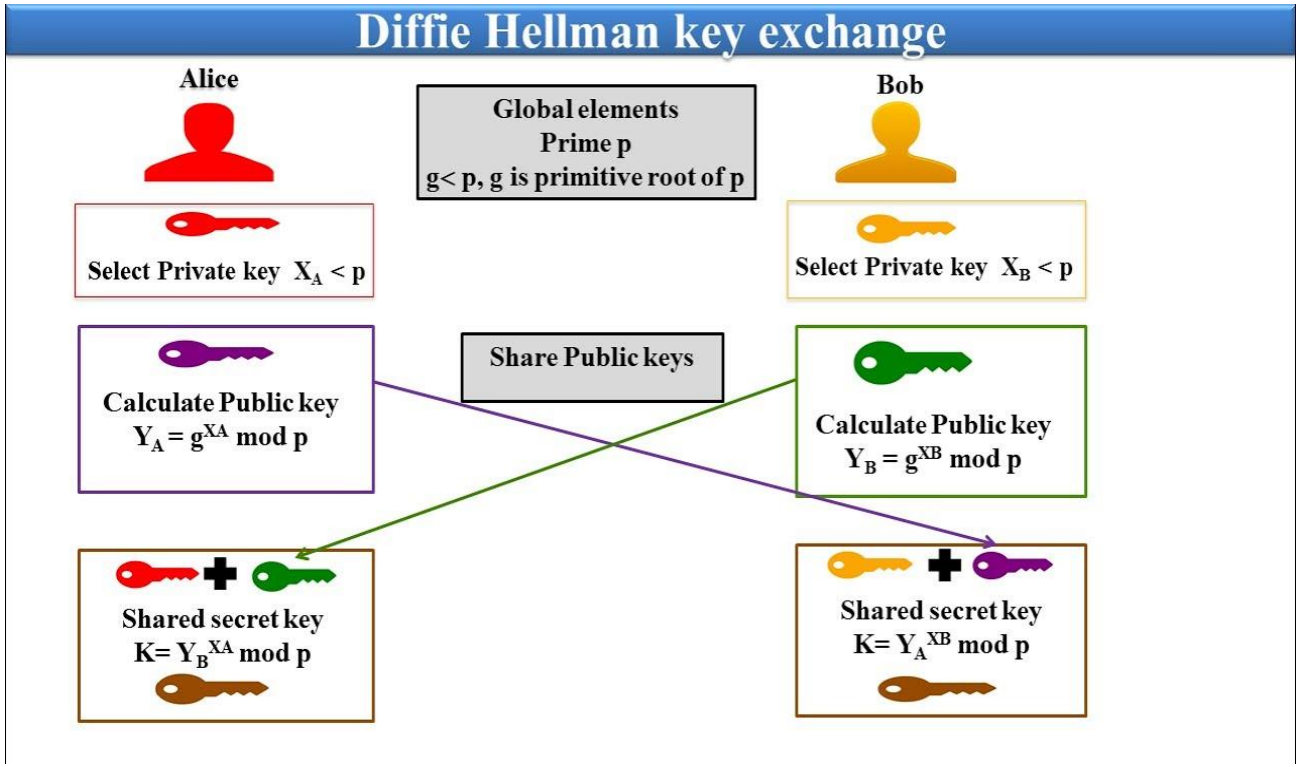


Steps in device authentication

- Key exchange
- Signature signing/verification
- Session key generation
- Symmetric encryption

Key exchange

- Key exchange (also key establishment) is a method in cryptography by which cryptographic keys are exchanged between two parties, allowing use of a cryptographic algorithm.
- The two most popular key exchange algorithms are RSA and Diffie-Hellman.



Key exchange implementation

- Generate key pair for key exchange.

```
// declare variables for generate key pair
EVP_PKEY_CTX *paramGenCtx = NULL, *keyGenCtx = NULL;
EVP_PKEY *params = NULL, *keyPair = NULL;

paramGenCtx = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);

// init param context for generation
if (!EVP_PKEY_paramgen_init(paramGenCtx))
{
    freePointers_GenerateKeyPair(paramGenCtx, keyGenCtx);
    setErrorCode(GENERATE_KEY_PAIR_PARAM_INIT_ERR);
    printf("[ecdh_GenerateKeyPair] paramgen init failed.");
    return RET_FAILED;
}
```

```
// set ec domain type for param context
EVP_PKEY_CTX_set_ec_paramgen_curve_nid(paramGenCtx, KEY_EXCHANGE_DOMAIN);
EVP_PKEY_paramgen(paramGenCtx, &params);
keyGenCtx = EVP_PKEY_CTX_new(params, NULL);

// init keygen context
if (!EVP_PKEY_keygen_init(keyGenCtx))
{
    freePointers_GenerateKeyPair(paramGenCtx, keyGenCtx);
    setErrorCode(GENERATE_KEY_PAIR_KEYGEN_INIT_ERR);
    printf("[ecdh_GenerateKeyPair] keygen init failed.");
    return RET_FAILED;
}
```

```
// generate key pair
if (!EVP_PKEY_keygen(keyGenCtx, &keyPair))
{
    freePointers_GenerateKeyPair(paramGenCtx, keyGenCtx);
    setErrorCode(GENERATE_KEY_PAIR_KEYGEN_ERR);
    printf("[ecdh_GenerateKeyPair] key generate failed.");
    return RET_FAILED;
}
```

```
// get ec key from key pair
EC_KEY *ecKey = EVP_PKEY_get1_EC_KEY(keyPair);

// free pointers for key gen
freePointers_GenerateKeyPair(paramGenCtx, keyGenCtx);

// get private key in BIGNUM format
const BIGNUM *privKey = EC_KEY_get0_private_key(ecKey);

// get public point in EC_POINT format
const EC_POINT *pubPoint = EC_KEY_get0_public_key(ecKey);
```

Signature signing

- Sender will generate signature using own private key(secure key) which is saved in secured memory.

```
// set private key to sign
BIGNUM *priv = BN_new();
EC_KEY *privKey = EC_KEY_new_by_curve_name(KEY_EXCHANGE_DOMAIN);
BN_bin2bn(privateKey, ECDH_KEY_LENGTH, priv);
EC_KEY_set_private_key(privKey, priv);
```

```
// calculate hash for public key
int hash_len = 0;
uint8_t hash[ECDH_KEY_LENGTH] = {0};
hashCalculation(hash, &hash_len, publickey, ECDH_KEY_LENGTH * 2, INDEX_SHA_256);

// generate signature
ECDSA_SIG *signature = ECDSA_do_sign(hash, ECDH_KEY_LENGTH * 2, privKey);

// extract signature to binary array
BN_bn2bin(signature->r, sign_r);
BN_bn2bin(signature->s, sign_s);
```

Signature verification

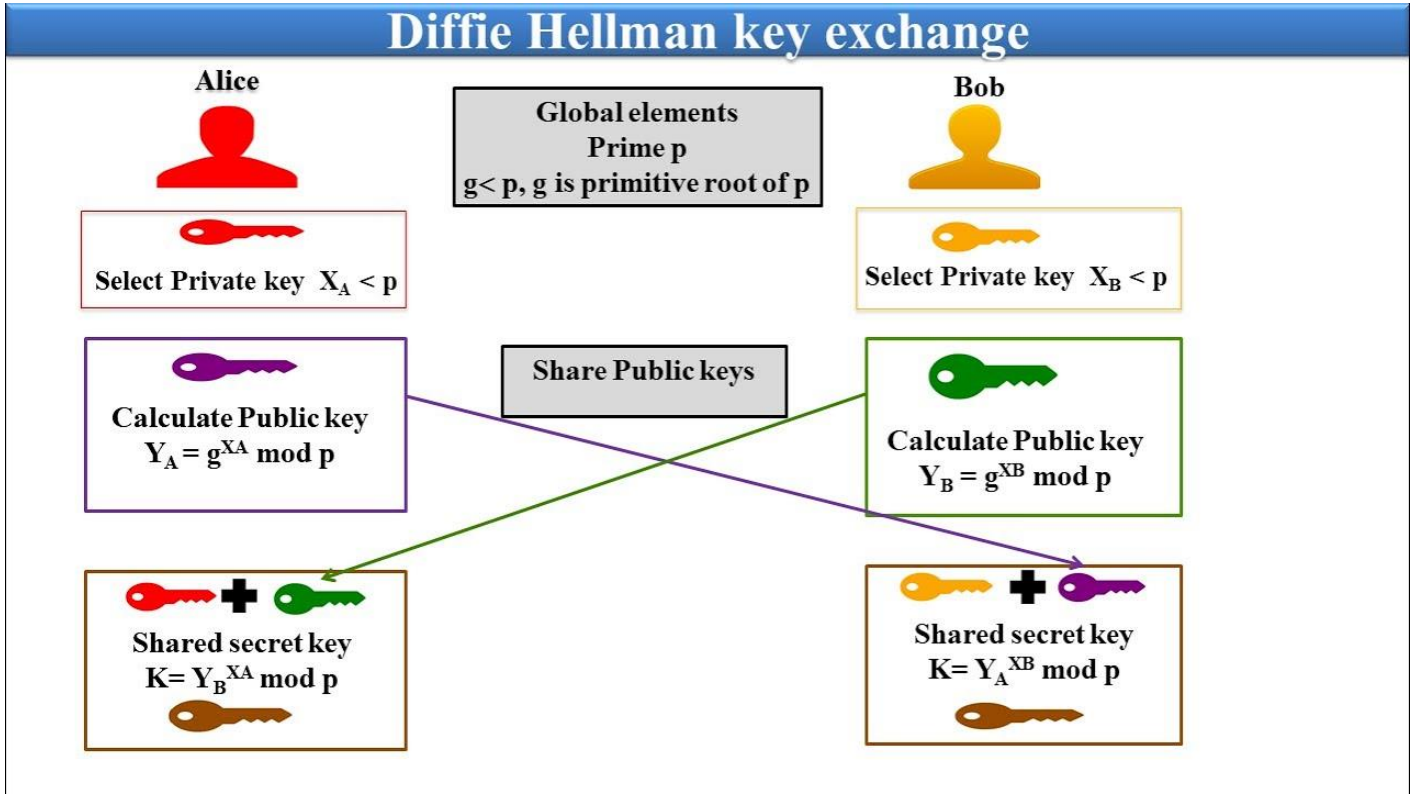
- Receiver will validate signature using public key which is saved in secured memory.

```
// convert public key from x, y to EC_KEY
EC_KEY *ec_pubKey = NULL;
EC_POINT *pub_key_point = NULL;
const EC_GROUP *group = NULL;
BIGNUM *x = BN_new();
BIGNUM *y = BN_new();
ec_pubKey = EC_KEY_new_by_curve_name(KEY_EXCHANGE_DOMAIN);
group = EC_KEY_get0_group(ec_pubKey);
pub_key_point = EC_POINT_new(group);
BN_bin2bn(publicXKey, ECDH_KEY_LENGTH, x);
BN_bin2bn(publicYKey, ECDH_KEY_LENGTH, y);
if (EC_POINT_set_affine_coordinates_GFp(group, pub_key_point, x, y, NULL) == RET_FAILED)
{
    return false;
}
EC_KEY_set_public_key(ec_pubKey, pub_key_point);
if (!EC_KEY_check_key(ec_pubKey))
{
    printf("EC_KEY_check_key failed.\n");
    return false;
}
```

```
// verify signature
BIGNUM *r = BN_new();
BIGNUM *s = BN_new();
BN_bin2bn(sign_r, ECDH_KEY_LENGTH, r);
BN_bin2bn(sign_s, ECDH_KEY_LENGTH, s);
ECDSA_SIG *signature = ECDSA_SIG_new();
signature->r = r;
signature->s = s;
if (ECDSA_do_verify(hash, hash_len, signature, ec_pubKey) != 1) {
    printf("Client verify signature OK.\n\n");
} else {
    printf("Client verify signature failed.\n\n");
    return false;
}
```

Session key generation – Create share secret key

- From pairing of public key and private key, we can obtain share secret key.



Session key generation – Create share secret key

```
BN_bin2bn(publicKeyX, publicKeyXLength, x);
BN_bin2bn(publicKeyY, publicKeyYLength, y);
BN_bin2bn(privateKey, privateKeyLength, priv);

// create private key and set priv number
ec_privateKey = EC_KEY_new_by_curve_name(KEY_EXCHANGE_DOMAIN);
if (!EC_KEY_set_private_key(ec_privateKey, priv))
{
    // EC_KEY_set_private_key failed
    freePointers_GenerateSharedSecretKey(x, y, priv, ec_privateKey, group, pubPoint);
    setErrorCode(GENERATE_SHARE_SECRET_PRIVATE_KEY_ERR);
    printf("[ecdh_GenerateSharedSecretKey] set private key failed.");
    return RET_FAILED;
}
```

```
// create group
group = EC_GROUP_new_by_curve_name(KEY_EXCHANGE_DOMAIN);

// create public point
pubPoint = EC_POINT_new(group);
if (!EC_POINT_set_affine_coordinates_GFp(group, pubPoint, x, y, NULL))
{
    // EC_POINT_set_affine_coordinates_GFp failed.
    freePointers_GenerateSharedSecretKey(x, y, priv, ec_privateKey, group, pubPoint);
    setErrorCode(GENERATE_SHARE_SECRET_PUBLIC_KEY_ERR);
    printf("[ecdh_GenerateSharedSecretKey] set public point failed.");
    return RET_FAILED;
}
```

```
// compute share secret key
int ret = ECDH_compute_key(shareSecret, *shareSecretLength, pubPoint, ec_privateKey, NULL);
*shareSecretLength = ret;
```

Session key generation

- Session key will be generate from share secret key by using KDF algorithm.

```
6. For i = 1 to reps, do the following:
6.1 Increment counter by 1.
6.2 Compute K(i) = H(counter || Z || FixedInfo).
6.3 Set Result(i) = Result(i – 1) ||K(i).
*/
int index = 1;
for (; index <= resp; ++index)
{
    /* counter has the same value as index but is always in a big endian form */
    memset(counter, 0, COUNTER_SIZE_IN_BYTES * sizeof(uchar_t));
    counter[3] = index & 0xFFu;
    counter[2] = (index >> 8) & 0xFFu;
    counter[1] = (index >> 16) & 0xFFu;
    counter[0] = (index >> 24) & 0xFFu;

    memset(inputBuff, 0, input_size * sizeof(uchar_t));
    // concat counter || Z || fix info
    memcpy(inputBuff, counter, COUNTER_SIZE_IN_BYTES);
    memcpy(inputBuff + COUNTER_SIZE_IN_BYTES, shareSecret, shareSecretLength);
    memcpy(inputBuff + COUNTER_SIZE_IN_BYTES + shareSecretLength, fixInfo, fixInfoLength);

    // calculate hash for input buffer
    uchar_t *output = hashAlgorithm->hash_function(inputBuff, input_size, NULL);
    if (NULL == output)
    {
        // free input buffer
        free(inputBuff);
        setErrorCode(KDF_HASH_ERR);
        printf("[kdf_process] hash calculate failed.");
        return RET_FAILED;
    }

    // set output of session key length
    memcpy(keyDerived + (index - 1) * hashAlgorithm->hashsize, output, keyDerivedLength - (index - 1) * hashAlgorithm->hashsize);
}
```

Symmetric encryption – AES algorithm

```
// create aes key
AES_KEY aes_key = {0x00};
if (0 != AES_set_encrypt_key(sessionKey, sessionKeyLen * BIT_PER_BYTE, &aes_key))
{
    setErrorCode(ENCRYPT_RANDOM_KEY_ERR);
    printf("[aesecb_EncryptRandomNumber] failed to set encrypt key.");
    return RET_FAILED;
}
```

```
// init length of encrypted data
*encryptDataLen = 0;
int num_block = ceil((float)randomNumberLen / (float)AES_BLOCK_SIZE);
int index = 0;
int pos = 0;
while (index < num_block)
{
    // encrypt block data
    if (pos + AES_BLOCK_SIZE <= randomNumberLen)
    {
        AES_encrypt(randomNumber + pos, encryptData + pos, &aes_key);
    }
    else // encrypt last block data
    {
        const uchar_t pad_data[AES_BLOCK_SIZE] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10};
        uchar_t tmp_data[AES_BLOCK_SIZE] = {0x00};
        memcpy(tmp_data, randomNumber + pos, randomNumberLen - pos);
        memcpy(tmp_data + randomNumberLen - pos, pad_data, pos + AES_BLOCK_SIZE - randomNumberLen);
        AES_encrypt(tmp_data, &encryptData[pos], &aes_key);
    }

    // increase encrypted data length
    pos += AES_BLOCK_SIZE;
    ++index;
}
```


Q
A


THANK YOU